ADLINK

GPB Tutorial

Release 2.0.14

Contents

1

Preface

1.1 About the Vortex Lite Google Protocol Buffers Tutorial

This *Vortex Lite GPB Tutorial* is included with the Vortex Lite DDS Documentation Set.

It describes how to use the Vortex Lite **ISO C++ API** in combination with **Google Protocol Buffers** (*GPB*) data models.

This Tutorial assumes that the user has installed Google Protocol buffer package & is already familiar with the DDS API as well as the Vortex Lite product.

Intended Audience

This Guide is intended for anyone who wants to use **Google Protocol Buffers for DDS** in developing and running applications with Vortex Lite.

1.2 Conventions

The icons shown below are used to help readers to quickly identify information relevant to their specific use of Vortex Lite.

Icon	*Meaning*
	Item of special significance or where caution needs to be taken.
	Item contains helpful hint or special information.

2

Introduction

2.1 Google Protocol Buffers for DDS

Vortex Lite is capable of using the **Google Protocol Buffer** (*GPB*) system for publishing and subscribing GPB messages in a DDS system. This makes it possible to use GPB as an alternative to OMG-IDL for those who prefer to use GPB rather than IDL. With the seamless integration of GPB and DDS technologies there is no need for OMG-IDL knowledge or visibility when working with GPB data models, and no OMG-DDS data-types are needed in the application (no explicit type-mapping between GPB and DDS types is required).

This results in an easy migration of GPB users to DDS(-based data-sharing) with data-centric GPB with support for keys, filters and (future) QoS-annotations (ony a few DDS calls are needed). Also easy migration of DDS applications to GPB(-based data-modeling), only the field accessors change.

This Tutorial will describe how this is done for the language binding **ISO-C++** by defining a GPB message layout which is compiled into proper interfaces for the Vortex DDS system.

2.1.1 GPB Installation and usage with DDS

Google Protocol Buffers (GPB) can be downloaded from the following locations:

Linux: https://github.com/google/protobuf/releases/download/v2.6.1/protobuf-2.6.1.tar.gz

Windows: https://github.com/google/protobuf/releases/download/v2.6.1/protobuf-2.6.1.zip

After unpacking follow the install instructions located in install.txt in the unpacked directory. For windows, follow instructions from readme.txt in vsprojects directory that will build everything that is needed.

2.1.2 IDL usage in a DDS system

In a Data Distributed System (DDS) as a Global DataSpace (GDS) for ubiquitous information-sharing in distributed systems as specified by the Object Management Group (OMG), the data is traditionally captured in the platform- and language-independent OMG-IDL language. The relational model of DDS is supported by the notion of identifying key fields in these data structures where structure/content-awareness by the middleware allows for dynamic querying and filtering of data.

2.1.3 Google Protocol Buffers

Google Protocol Buffers (GPB) are a flexible, efficient, automated mechanism for serializing structured data; think XML, but smaller, faster, and simpler. One can define how data needs to be structured once, after which language-specific source code can be generated to easily write and read this structured data to and from a variety of data streams using a variety of languages. The information structure is defined in so-called *protocol buffer* message types in `.proto` files. Each protocol buffer message is a small logical record of information, containing a series of name-value pairs. This approach is quite similar to using IDL for data modeling in combination with an IDL compiler (as available in Vortex Lite and DDS implementations in general).

Additionally, the GPB data structure can be updated without breaking deployed programs that are compiled against the 'old' format, similar to the *xTypes* concept as defined for DDS.

Using a GPB data-model instead of an IDL data-model

For an IDL-OMG based application, the IDL file is compiled with the IDL-PP compiler to generate the needed classes.

For C++ as an example, `address.idl` will (among others) be compiled into:

- `address.cpp`
- `address_DCPS.hpp`
- `addressSplDcps.cpp`
- ...

Using a GPB data-model, it is not necessary to create IDL files. The `protoc_gen_ddsJava` plug-in in OpenSplice will create them from the given `.proto` data-model.

For the GPB `.proto` based application, the `.proto` file is first compiled by the Google `protoc` compiler. This compiler will call the `protoc_gen_ddslcpp` plug-in in Vortex Lite with the `.proto` data parsed into a `CodeGeneratorRequest` protocol buffer.

The Vortex Lite plug-in will generate an IDL file from this data. Any field member that is marked as key or filterable is explicitly mapped to a member in the IDL type.

The complete serialized `.proto` message is stored in the generic `ospl_protobuf_data` attribute as a sequence of bytes (making it opaque data for DDS). The mapping between data types is given in the table Mapping of GPB types to DDS types.

As the next step the IDL-PP compiler will generate the previously-named files from the idl file needed for the DDS domain. The Google `protoc` compiler will generate the classes needed for the GPB domain.

The dds options for the proto file are given in the omg/dds/descriptor.proto file listed below. This proto file shows how the different dds options on the proto file are interpreted, and gives the unique id `1016` to the dds types.

Note that the id 1016 has officially been granted to the Vortex product by Google.
This ensures these options are always unique and won't clash with any options used by users.

omg/dds/descriptor.proto

```
import "google/protobuf/descriptor.proto";

package omg.dds;

option java_package = "org.omg.dds.protobuf";
option java_outer_classname = "DescriptorProtos";

/* These options are required for any .proto message that needs to be available
 * in DDS.
 *
 * - name: An optional scoped name to allow overriding the name of the type in
 *   DDS. The dot('.') can be used as a scoping separator. In case the name
 *   starts with a dot, the name will be interpreted as an absolute scope name.
 *   If not, the name will be considered relative to the scope of the message
 *   including its 'package'.
 */
message MessageOptions {
    optional string name = 1 [default = ""];
}

extend google.protobuf.MessageOptions {
  optional omg.dds.MessageOptions type  = 1016;
}
```

```
/* These options are provided to assign specific behaviour to a member of a
 * DDS-enabled .proto message in DDS. These options will only be applied in case
 * the omg.dds.MessageOptions.type has been applied to the message in which the
 * member is modeled.
 *
 * - key: Make the member part of the key of the type in DDS. Each unique
 *   key-value will become a separate instance with its own history in DDS. Only
 *   'required' members can be made part of the key and key-definitions cannot
 *   be modified in future versions of the message. Members that are part of the
 *   key are automatically filterable as well.
 *
 * - filterable: Ensure the member is filterable in DDS using a so-called
 *   ContentFilteredTopic or QueryCondition. Only 'required' members can be made
 *   filterable and filterable definitions cannot be modified in future versions
 *   of the message.
 *
 * - name: Override the name of the member in DDS. This only applies to members
 *   that are marked as key and/or filterable.
 */
message FieldOptions {
    optional bool key = 1 [default = false];
    optional bool filterable = 2 [default = false];
    optional string name = 3 [default = ""];
}

extend google.protobuf.FieldOptions {
  optional omg.dds.FieldOptions member = 1016;
}
```

How mapping is done between the different languages is shown below in the table Mapping of GPB types to DDS types.

Mapping of GPB types to DDS types

.proto Type	Notes	C++ Type	Java Type	DDS IDL Type
double		double	double	double
float		float	float	float
int32	Uses variable-length encoding. Inefficient for encoding negative numbers; if your field is likely to have negative values, use sint32 instead	int32	int	long
int64	Uses variable-length encoding. Inefficient for encoding negative numbers; if your field is likely to have negative values, use sint64 instead	int64	long	long long
uint32	Uses variable-length encoding	uint32	int	unsigned long
uint64	Uses variable-length encoding	uint64	long	unsigned long long
sint32	Uses variable-length encoding. Signed int value. These encode negative numbers more efficiently than regular int32s.	int32	int	long
sint64	Uses variable-length encoding. Signed int value. These encode negative numbers more efficiently than regular int64s.	int64	long	long long
fixed32	Always four bytes. More efficient than uint32 if values are often greater than 2^28.	uint32	int	unsigned long
fixed64	Always eight bytes. More efficient than uint64 if values are often greater than 2^56.	uint64	long	unsigned long long
sfixed32	Always four bytes.	int32	int	long
sfixed64	Always eight bytes.	int64	long	long long
bool		bool	boolean	bool
string	A string must always contain UTF-8 encoded or 7-bit ASCII text	string	String	string

3

Proto message for a DDS system

Individual declarations in a `.proto` file can be annotated with a number of options. Options do not change the overall meaning of a declaration, but may affect the way it is handled in a particular context.

Options can be defined at different levels:

- File-level options: meaning they should be written at the top-level scope, not inside any message, enum, or service definition.
- Message-level options: meaning they should be written inside message definitions.
- Field-level options: meaning they should be written inside field definitions. Enum types, enum values, service types, and service methods.

3.1 Use case: Person

In this use case example, a system capable of describing the personal data of persons must be built using the GPB data-model

The layout can be:

```
string name
integer age
sequence phone-number + type
sequence friends
```

3.2 Proto file for the Person example

This use case is described in this `.proto` file:

```
import "omg/dds/descriptor.proto";
package address;

message Person {
   required string name = 1;
   required int32 age = 2;
   optional string email = 3;

   enum PhoneType {
      UNDEFINED = 0;
      MOBILE    = 1;
      HOME      = 2;
      WORK      = 3;
   }

   message PhoneNumber {
      required string number = 1;
      optional PhoneType type  = 2 [default = HOME];
```

```
    }

repeated PhoneNumber phone = 4;
repeated Person friend = 5;
}
```

GPB labels every field as either a *required* or an *optional* field. Required fields are *always* used/filled; optional fields may or may not be.

Different data models are compatible if all *required* fields are the same. Data models can be extended with extra fields; if those new fields are all *optional*, then the new model will still be compatible with older applications using the old data model.

In our example the *name* and *age* are always required. The *email* string is optional, as extra information for this person. The sequences with phone numbers and friends are allowed to be empty.

Detailed explanation for the layout of a `.proto` file can be found in the Google Protocol buffer documentation on https://developers.google.com/protocol-buffers/docs/proto

3.3 Annotating a proto message for use as a type in DDS

For the GPB message to be able to be handled correctly in a DDS system, some options are needed in the `.proto` file which define how the GPB message shall behave in the DDS system.

At the message level there is an extra option `.omg.dds.type`. This tells the protocol buffer compiler that this message is also a dds type message. This type option has a optional extra parameter for giving this type a dds type name. By default it has the same name in the DDS domain as it has in GPB.

The Person example with this option:

```
import "omg/dds/descriptor.proto";

package address;

message Person {
 option (.omg.dds.type) = {};
 required string name = 1;
 required int32 age = 2;

enum PhoneType {
  UNDEFINED = 0;
  MOBILE    = 1;
  HOME      = 2;
  WORK      = 3;
}

message PhoneNumber {
   required string number = 1;
  optional PhoneType type  = 2 [default = HOME];
 }
 repeated PhoneNumber phone = 4;
 repeated Person friend = 5;
}
```

3.3.1 Proto file with omg.dds.member.key option

For support of a key value in the datamodel, the option `key` can be given as a field-member option. One or more fields containing this option will indicate that these members make a unique key identifier in the data model. A field indicated as a key field must always be a *required* field for GPB. Also a key field is automatically a filterable field, as described below.

The Person example with *name* as a unique key (this means that each unique value of the name will lead to a separate instance in DDS with its own history):

```
import "omg/dds/descriptor.proto";

package address;

message Person {
   option (.omg.dds.type) = {};
   required string name = 1 [(.omg.dds.member).key = true];
   required int32 age = 2;
   optional string email = 3;
}
enum PhoneType {
   UNDEFINED = 0;
   MOBILE = 1;
   HOME = 2;
   WORK = 3;
}
message PhoneNumber {
   required string number = 1;
   optional PhoneType type = 2 [default = HOME];
}

repeated PhoneNumber phone = 4;
repeated Person friend = 5;
```

3.3.2 Proto file with omg.dds.member.filterable option

For support of filterable fields in the datamodel, the option `filterable` can be given as a field-member option.

One or more fields with this option indicates that these members are available for dynamic querying and filtering by means of a `QueryCondition` or `ContentFilteredTopic` in DDS.

A field marked as a filterable field must always be a *required* field in GPB. A key field is always filterable, by definition.

The Person example with *age* as a filterable attribute:

```
import "omg/dds/descriptor.proto";

package address;

message Person {
 option (.omg.dds.type) = {};
 required string name = 1 [(.omg.dds.member).key = true];
 required int32 age = 2 [(.omg.dds.member).filterable = true];
 optional string email = 3;

enum PhoneType {
  UNDEFINED = 0;
  MOBILE    = 1;
  HOME      = 2;
  WORK      = 3;
}

message PhoneNumber {
   required string number = 1;
  optional PhoneType type  = 2 [default = HOME];
 }
 repeated PhoneNumber phone = 4;
 repeated Person friend = 5;
}
```

3.3.3 Proto file with omg.dds.member.name option

The previous examples will result in a DDS type with the directly-mapped fields in IDL with the same name as in proto. (Key fields and filterable fields are directly mapped.)

If a different name is needed in the DDS domain for a fieldname in the generated IDL and dds type, a name can be given as an `omg.dds.member` option.

Example where the *age* field will be named `AgeInYears` in the DDS domain:

```
import "omg/dds/descriptor.proto";

package address;

message Person {
 option (.omg.dds.type) = {};
 required string name = 1 [(.omg.dds.member).key = true];
 required int32 age = 2   [(.omg.dds.member) = { name: "AgeInYears" filterable: true }];
 optional string email = 3 ;

enum PhoneType {
  UNDEFINED = 0;
  MOBILE    = 1;
  HOME      = 2;
  WORK      = 3;
}

message PhoneNumber {
   required string number = 1;
  optional PhoneType type  = 2 [default = HOME];
 }
 repeated PhoneNumber phone = 4;
 repeated Person friend = 5;
}
```

4

Compiling the datamodel with the GPB compiler

Once you've defined your messages, you run the protocol buffer compiler for your application's language on your `.proto` file to generate data access classes. These provide simple accessors for each field so, for instance, if your chosen language is ISO-C++, running the compiler on the above example will generate a class called *Person*. You can then use this class in your application to populate, serialize, and retrieve *Person* protocol buffer messages.

4.1 DDS-specific GPB-compiler plugin to generate code.

The GPB compiler can be extended to support new languages *via* so-called plugins. The compiler invokes the plugin while providing the GPB type definition to it in the form of a GPB message. For DDS support the Vortex Lite GPB-compiler is delivered with Vortex Lite.

The Vortex Lite IDL compiler is invoked by the Lite GPB-compiler plugin to generate the DDS type including typed DataWriter and DataReader code. Additionally, code is generated to convert an instance of the DDS type to the GPB type and *vice versa*, which hides the DDS type from the application entirely.

4.2 C++ example

For creating the DDS specific code by the GPB compiler the option –ddslcpp_out must be given to the compiler. Also the path to the Vortex Lite GPB-compiler must be given. Example:

```
protoc --cpp_out     =outputPath
       --ddslcpp_out  =outputPath
       --proto_path  =PathToProtoFile
       --proto_path  =PathToProtofile
       protoFileToCompile
```

- `--cpp_out` gives the path where the GDP generated code will be stored.
- `--ddscpp_out` gives the path where the DDS-specific generated code will be stored.
- first `--proto_path`: the protoc compiler needs the path where the `.proto` file is located.
- second `--proto_path`: the path where the GPB depencies are stored
- `protoFileToCompile` the last option is the `.proto` file.

Assuming that we need the generated code in the current directory and the previous `address.proto` example is in the current directory, the command will be:

```
protoc --cpp_out=.
       --ddslcpp_out=.
       --proto_path=.
       --proto_path=$LITE_HOME/etc/protobuf
       ./address.proto
```

The generated code, in the current directory, can be compiled normally with the C++ compiler together with your own written applications.

This example is delivered with Vortex Lite, and is located in `examples/isocpp/protobuf`.

If the generated `.idl` file is needed by other applications, this file will also be generated in the `--ddslcpp_out` path if the environment variable `LITE_PROTOBUF_INCLUDE_IDL` is set to true.

4.3 Tempory IDL file created by the GPB data-model

The IDL file created for the previous example will contain:

```
module org {
   module omg {
      module dds {
         module protobuf {
            typedef sequence<octet> gpb_payload_t;
         };
      };
   };
};

module address {
    module dds {
        struct Person {
            string name;
            long age;
            string worksFor_name;
            string worksFor_address;
            ::org::omg::dds::protobuf::gpb_payload_t ospl_protobuf_data;
        };
        #pragma keylist Person name worksFor_name
    };
};
```

This idl file is deleted after the idl-pp compiler is finished. If the temporary idl file is needed in other DDS applications (it also usable for other DDS vendors), then the environment variable `LITE_PROTOBUF_INCLUDE_IDL` must be set to true to prevent the idl file from being deleted.

5

Using the generated API in applications

The DDS API implementation will allow the use of GPB types for DDS transparently, and the generated underlying DDS type will be invisible to the application.

5.1 ISO-C++

In this example the publisher and subscriber are embedded into one file.

The publisher part will publish a person *Jane Doe* with one friend, *John Doe*.

The Subscriber part in this example will read this data and print it to the `stdout`.

This example is delivered with Vortex Lite, and is located in `examples/isocpp/protobuf`.

```
/*
 *                         Vortex Lite
 *
 *   This software and documentation are Copyright 2006 to 2017 ADLINK
 *   Technology Limited, its affiliated companies and licensors.
 *   All rights reserved. See file:
 *
 *                     $LITE_HOME/LICENSE
 *
 *   for full copyright notice and license terms.
 *
 */

#include "implementation.hpp"
#include "cpp/utils/example_utilities.h"

#include <iostream>

#include "address.pbdds.hpp"

namespace examples { namespace protobuf { namespace isocpp  {

int publisher(int argc, char *argv[])
{
    int result = 0;
    (void) argc;
    (void) argv;
    try
    {

      std::cout << "here!!!... " <<std::endl;
      /** A dds::domain::DomainParticipant is created for the default domain. */
      dds::domain::DomainParticipant dp(org::opensplice::domain::default_id());

      /** A dds::topic::Topic is created for our protobuf type on the domain
          participant. */
```

```
    dds::topic::Topic<address::Person> topic(dp, "Person");

    /** A dds::pub::Publisher is created on the domain participant. */
    dds::pub::Publisher pub(dp);

    /** The dds::pub::qos::DataWriterQos is derived from the topic qos */
    dds::pub::qos::DataWriterQos dwqos;

    dwqos << dds::core::policy::Reliability::Reliable();
    /** A dds::pub::DataWriter is created on the Publisher & Topic with
        the modififed Qos. */
    dds::pub::DataWriter<address::Person> dw(pub, topic);

    /** Synchronize on subscriber availability. */
    std::cout << "Publisher: waiting for subscriber... " <<std::endl;
    unsigned long current = exampleTimevalToMicroseconds(exampleGetTime());
    unsigned long timeout = current + (30 * 1000 * 1000);
    bool stop = false;
    ::dds::core::status::PublicationMatchedStatus matched;

    do {
         matched = dw.publication_matched_status();

         if (exampleTimevalToMicroseconds(exampleGetTime()) > timeout) {
             stop = true;
         }
         if ((matched.current_count() == 0) && (!stop)) {
             exampleSleepMilliseconds(500);
         }
      } while ((matched.current_count() == 0) && (!stop));

      if (matched.current_count() != 0) {
          std::cout << "Publisher: Subscriber found" << std::endl;

      /** A sample is created and then written. */
      address::Person msgInstance;
      msgInstance.set_name("Jane Doe");
      msgInstance.set_email("jane.doe@somedomain.com");
      msgInstance.set_age(23);

      address::Person::PhoneNumber* phone = msgInstance.add_phone();
      phone->set_number("0123456789");
      address::Organisation* worksFor = msgInstance.mutable_worksfor();
      worksFor->set_name("Acme Corporation");
      worksFor->set_address("Wayne Manor, Gotham City");
      worksFor->mutable_phone()->set_number("9876543210");
      worksFor->mutable_phone()->
      set_type( ::address::Person_PhoneType_WORK);
      std::cout << "Publisher: publishing Person: " <<
      msgInstance.name() << std::endl;

      ::dds::core::InstanceHandle handle = dw.register_instance(msgInstance);
       dw << msgInstance;

       std::cout << "Publisher: sleeping for 5 seconds..." << std::endl;

       exampleSleepMilliseconds(5000);

       std::cout << "Publisher: disposing Jane Doe..." << std::endl;

       /** Disposing the DDS instance associated with the name field of the
           * Protobuf data structure which is the key in DDS*/
```

```
            dw.dispose_instance(handle);
        } else {
            throw ::dds::core::PreconditionNotMetError
            ("Subscriber NOT found, terminating...");
        }
    }
    catch (const dds::core::Exception& e)
    {
        std::cerr << "Publisher: ERROR: " << e.what() << std::endl;
        result = 1;
    }
    std::cout << "Publisher: terminating..." << std::endl;
    return result;
}

class ReadCondHandler
{
public:
    /**
     * @param dataState The dataState on which to filter the samples
     */
    ReadCondHandler(): updateCount(0) {}

    void operator() (dds::sub::cond::ReadCondition c)
    {
        std::string states, sampleState, viewState, instanceState;
        dds::sub::DataReader<address::Person> dr = c.data_reader();
        dds::sub::LoanedSamples<address::Person> samples =
        dr.select().state(c.state_filter()).take();

        for (dds::sub::LoanedSamples<address::Person>::const_iterator sample =
            samples.begin(); sample < samples.end(); ++sample)
        {
          updateCount++;

          if(sample->info().state().sample_state() ==
             dds::sub::status::SampleState::read())
          {
            sampleState = "READ";
          }
          else
          {
            sampleState = "NOT_READ";
          }
          if(sample->info().state().view_state() ==
            dds::sub::status::ViewState::new_view())
          {
            viewState = "NEW";
          }
          else
          {
            viewState = "NOT_NEW";
          }
          if(sample->info().state().instance_state() ==
            dds::sub::status::InstanceState::alive())
          {
            instanceState = "ALIVE";
          }
          else if(sample->info().state().instance_state() ==
            dds::sub::status::InstanceState::not_alive_disposed())
          {
            instanceState = "NOT_ALIVE_DISPOSED";
          }
```

```
          else
          {
            instanceState = "NOT_ALIVE_NO_WRITERS";
          }
          states =
          "(" + sampleState + ", " + viewState + ", " + instanceState + ")";

          if(sample->info().valid())
          {
            std::cout << "Subscriber: reading sample " <<  states << ":" <<
            std::endl;
            printPerson(sample->data(), "");
          }
          else
          {
            std::cout << "Subscriber: reading invalid sample " << states << ":" <<
            std::endl;
            std::cout << "- Name  = " << sample->data().name() << std::endl;
          }
        }
    }

    int getUpdateCount(){
        return updateCount;
    }

private:
  int updateCount;

  void printPhone(const ::address::Person_PhoneNumber phone, std::string tabs){
      std::string type;

      switch(phone.type()){
      case ::address::Person_PhoneType_MOBILE:
          type = "MOBILE";
          break;
      case ::address::Person_PhoneType_HOME:
          type = "HOME";
          break;
      case ::address::Person_PhoneType_WORK:
          type = "WORK";
          break;
      default:
          type = "UNKNOWN";
          break;
      }
      std::cout << tabs << "- Phone      = " << phone.number() <<
              " (" + type << ")" << std::endl;

    }

    void printPerson(address::Person person, std::string tabs){
        std::cout << tabs << "- Name       = " << person.name() << std::endl;
        std::cout << tabs << "- Age        = " << person.age() << std::endl;
        std::cout << tabs << "- Email      = " << person.email() << std::endl;

        for (int i=0; i< person.phone_size(); i++) {
            printPhone(person.phone(i), tabs);
        }
        std::cout << tabs << "- Company    = " << std::endl;
        std::cout << tabs << "    - Name       = " << person.worksfor().name() <<
        std::endl;
        std::cout << tabs << "    - Address    = " << person.worksfor().address() <<
```

```
        std::endl;

        if(person.worksfor().has_phone()){
            printPhone(person.worksfor().phone(), tabs + "   ");
        } else {
            std::cout << tabs << "   - Phone   = NONE" << std::endl;
        }
    }
};


/**
 * Runs the subscriber role of this example.
 * @return 0 if a sample is successfully read, 1 otherwise.
 */
int subscriber(int argc, char *argv[])
{
    int result = 0;
    (void) argc;
    (void) argv;
    try
    {
      int expectedUpdates = 2;
      /** A dds::domain::DomainParticipant is created for the default domain.*/
      dds::domain::DomainParticipant dp(org::opensplice::domain::default_id());

      /** A dds::topic::Topic is created for our protobuf type on the
          domain participant. */
      dds::topic::Topic<address::Person> topic(dp, "Person");

      /** A dds::pub::Subscriber is created on the domain participant. */
      dds::sub::Subscriber sub(dp);

      /** The dds::pub::qos::DataWriterQos is derived from the topic qos */
      dds::sub::qos::DataReaderQos drqos;

      drqos << dds::core::policy::Reliability::Reliable();
      /** A dds::pub::Reader is created on the Subscriber & Topic with the
          modified Qos. */
      dds::sub::DataReader<address::Person> dr =
      dds::sub::DataReader<address::Person>(sub, topic, drqos);

      /** any sample, view and instance state */
      dds::sub::cond::ReadCondition
        readCond(dr, dds::sub::status::DataState::any());
      ReadCondHandler personHandler;
      readCond.handler(personHandler);

     /** A WaitSet is created and the four conditions created
         above are attached to it */
     dds::core::cond::WaitSet waitSet;
     waitSet += readCond;

     dds::core::Duration waitTimeout(30, 0);

     /** Wait until the condition in the WaitSet triggers and
         dispatch the corresponding functor*/
     do {
          waitSet.dispatch(waitTimeout);
        } while(personHandler.getUpdateCount() < expectedUpdates);
    }
    catch (const dds::core::Exception& e)
    {
```

```
        std::cerr << "Subscriber: ERROR: " << e.what() << std::endl;
        result = 1;
    }
    std::cout << "Subscriber: terminating..." << std::endl;

    return result;
}

}
}
}

EXAMPLE_ENTRYPOINT(DCPS_ISOCPP_Protobuf_publisher, examples::protobuf::isocpp::publisher)
EXAMPLE_ENTRYPOINT(DCPS_ISOCPP_Protobuf_subscriber, examples::protobuf::isocpp::subscriber)
```

6

Evolving data models

It is likely that over time a data model will change; for example, new fields with extra information are often added to an existing data model.

Normally all applications using that data model need to be recompiled against the new (changed) data model in order to be aware of the extra fields. However, when using a data model based on the GPB system, it is possible to add extra fields to the data model and use applications based on the original data model *and* applications based on the new data model in a mixed environment.

It is only possible to combine old and new applications with different data models as long as the required fields are the same. Remember, a key field or a filterable field is always required, so these fields can not be changed or added if it is necessary to combine old and new data models.

In our example we will make a new data model with some extra fields:

- new phonetype: *SKYPE*
- new phoneNumber property: *secret*
- new string for an alias: *facebookname*

Proto file with new options:

```
import "omg/dds/descriptor.proto";
package address;
message Person {
   option (.omg.dds.type); // default type-name will be 'Person'
   required string name = 1 [(.omg.dds.member).key = true];
   required int32 age = 2   [(.omg.dds.member).filterable = true];
   optional string email = 3;

     enum PhoneType {
       UNDEFINED = 0;
       MOBILE = 1;
       HOME = 2;
       WORK = 3;
       SKYPE = 4; // **** added SKYPE phonetype enum-value ****
     }
     message PhoneNumber {
       required string number = 1;
       optional PhoneType type = 2 [default = UNDEFINED];
       optional bool secret = 3 [default = false];
        // **** added a new phoneNumber property ****
     }
  repeated PhoneNumber phone = 4;
  repeated Person friend = 5;
  optional string facebookname = 6 [default = "NONE"];
   // **** added alias facebook name ****
}
```

6.1 Old publisher and old subscriber

Data printed by subscriber program:

```
Name  = Jane Doe
Age   = 23
Email = jane.doe@somedomain.com
Phone = 0123456789 (HOME)
Friend:
   Name  = John Doe
   Age   = 35
   Email = john.doe@somedomain.com
```

6.2 New publisher and new subscriber

Data printed by subscriber program:

```
Name     = Jane Doe
Facebook = Jane123
Age      = 23
Email    = jane.doe@somedomain.com
Phone    = 0123456789 (HOME) secret=false
Phone    = 0612345678 (MOBILE) secret=true
Phone    = splicer (SKYPE) secret=false
Friend:
   Name     = John Doe
   Facebook = NONE
   Age      = 35
   Email    = john.doe@somedomain.com
```

6.3 Old publisher and new subscriber

New subscriber gets default values for absent fields:

```
Name     = Jane Doe
Facebook = NONE
Age      = 23
Email    = jane.doe@somedomain.com
Phone    = 0123456789 (HOME) secret=false
Friend:
   Name     = John Doe
   Facebook = NONE
   Age      = 35
   Email    = john.doe@somedomain.com
```

6.4 New publisher and old subscriber

Old subscriber doesn't understand the `SKYPE` phonetype so reverts to the default `UNDEFINED` phonetype.

Read data in old subscriber:

```
Name  = Jane Doe
Age   = 23
Email = jane.doe@somedomain.com
Phone = 0123456789 (HOME)
Phone = 0612345678 (MOBILE)
```

```
Phone = splicer (UNDEFINED)
Friend:
   Name  = John Doe
   Age   = 35
   Email = john.doe@somedomain.com
```

7

Contacts & Notices

7.1 Contacts

ADLINK Technology Corporation
400 TradeCenter
Suite 5900
Woburn, MA
01801
USA
Tel: +1 781 569 5819

ADLINK Technology Limited
The Edge
5th Avenue
Team Valley
Gateshead
NE11 0XA
UK
Tel: +44 (0)191 497 9900

ADLINK Technology SARL
28 rue Jean Rostand
91400 Orsay
France
Tel: +33 (1) 69 015354

Web: http://ist.adlinktech.com/

Contact: http://ist.adlinktech.com

E-mail: ist_info@adlinktech.com

LinkedIn: https://www.linkedin.com/company/79111/

Twitter: https://twitter.com/ADLINKTech_usa

Facebook: https://www.facebook.com/ADLINKTECH

7.2 Notices